**TradingView\_ProAI**

**19 June 2025**

Table of Contents

[Project Idea: TradingView Chart Pattern Classifier 2](#_Toc201236984)

[Project Breakdown (Step-by-Step): 2](#_Toc201236985)

[Phase 1: Dataset Preparation (Crucial and Most Time-Consuming) 2](#_Toc201236986)

[Phase 2: Python Project Setup (Similar to previous project) 4](#_Toc201236987)

[Phase 3: Python Code Implementation 5](#_Toc201236988)

# Project Idea: TradingView Chart Pattern Classifier

**Goal:** Develop a deep learning model to classify common chart patterns (e.g., "Uptrend," "Downtrend," "Sideways/Range," "Head & Shoulders," "Double Top/Bottom") from TradingView candlestick chart images.

**Why this project is interesting:**

* **Computer Vision Application:** You'll apply image classification techniques to a new domain.
* **Feature Learning:** The model will learn to identify visual features that define chart patterns.
* **Data Challenges:** Dealing with varying chart scales, timeframes, and visual styles.
* **Prediction (Indirect):** While not predicting price, classifying a pattern *implies* a likely future direction based on technical analysis principles (e.g., "Uptrend" suggests continued rise).

# Project Breakdown (Step-by-Step):

## Phase 1: Dataset Preparation (Crucial and Most Time-Consuming)

**Step 1: Define Your Chart Patterns (Classes)** Decide which patterns you want your model to recognize. Start simple (2-3 patterns) and expand later.

* **Example Classes:**
  + Uptrend
  + Downtrend
  + Sideways/Range
  + Strong Buy Signal (e.g., large green candlestick after a dip)
  + Strong Sell Signal (e.g., large red candlestick after a rise)
  + *(More Advanced: Head & Shoulders, Double Top, Double Bottom, Flags, Pennants - these are much harder to label consistently)*

**Step 2: Collect Images (102 Images from the Internet)** This is the manual part.

1. **Open TradingView (or similar charting platform):** You can use Bitcoin, Ethereum, S&P 500, or any other asset charts.
2. **Identify a Pattern:** Look at the chart and identify one of the patterns you defined in Step 1.
3. **Take a Screenshot:** Capture a screenshot of the chart **specifically showing that pattern**. Try to capture the pattern clearly, with some context around it.
4. **Save the Screenshot:** Save the image to your computer.
5. **Repeat 102 Times:** This will be tedious, but necessary for a small dataset. Try to get a relatively even distribution across your chosen classes.

**Step 3: Organize Your Dataset (Folder Structure and Labeling)** This is how your dataset should look on your computer:

trading\_chart\_dataset/

├── train/

│ ├── uptrend/

│ │ ├── uptrend\_001.png

│ │ ├── uptrend\_002.png

│ │ └── ...

│ ├── downtrend/

│ │ ├── downtrend\_001.png

│ │ ├── downtrend\_002.png

│ │ └── ...

│ └── sideways/

│ ├── sideways\_001.png

│ └── ...

├── val/

│ ├── uptrend/

│ │ ├── uptrend\_val\_001.png

│ │ └── ...

│ ├── downtrend/

│ │ ├── downtrend\_val\_001.png

│ │ └── ...

│ └── sideways/

│ ├── sideways\_val\_001.png

│ └── ...

└── test/

├── uptrend/

│ ├── uptrend\_test\_001.png

│ └── ...

├── downtrend/

│ ├── downtrend\_test\_001.png

│ └── ...

└── sideways/

├── sideways\_test\_001.png

└── ...

* **Total 102 Images:** Allocate them roughly 70-80% for train, 10-15% for val, and 10-15% for test. (e.g., 70-80 train, 10-15 val, 10-15 test).
* **Varying Sizes:** Don't worry too much about different screenshot sizes for now. Your data loading step will handle resizing.
* **Manual Labeling:** The folder name (uptrend, downtrend, etc.) acts as the label for all images within it. This is your ground truth.

**Step 4: Create a CSV for Dataset Information (Optional but Recommended)** You can create a dataset\_info.csv file manually or programmatically that lists the image path and its label. This makes it easier to manage and extend your dataset later.

Code snippet

image\_path,label

./train/uptrend/uptrend\_001.png,uptrend

./train/downtrend/downtrend\_001.png,downtrend

...

## Phase 2: Python Project Setup (Similar to previous project)

**Step 1: Project Folder Structure**

trading\_project/

├── main.py

├── dataset/

│ └── (your 'trading\_chart\_dataset' folder will go here)

├── src/

│ ├── dataset\_loader.py

│ ├── model\_architecture.py

│ └── utils.py

├── requirements.txt

**Step 2: requirements.txt** Create this file in the trading\_project/ directory and list your dependencies:

torch

torchvision

numpy

pandas

matplotlib

seaborn

scikit-learn

Pillow

tqdm

albumentations # If you want to use advanced image augmentations

opencv-python-headless # For image processing if needed (e.g., resizing)

**Step 3: Virtual Environment and Installation**

1. Open your terminal in VS Code within the trading\_project/ directory.
2. Create virtual environment: python -m venv venv
3. Activate:
   * Windows: .\venv\Scripts\activate
   * macOS/Linux: source venv/bin/activate
4. Install dependencies: pip install -r requirements.txt
   * **For Apple M1/M2/M3 (MPS):** Ensure you install PyTorch for MPS. The pip install torch torchvision torchaudio command should automatically pull the correct version if you are on a compatible macOS version with Apple Silicon. Verify with print(torch.backends.mps.is\_available()).

## Phase 3: Python Code Implementation

**1. src/dataset\_loader.py** This file will handle loading your images and applying transformations.

Python

# src/dataset\_loader.py

from torch.utils.data import Dataset, DataLoader

from torchvision import transforms # Use torchvision for common image transforms

from PIL import Image

import os

import pandas as pd # To load the CSV if you create one

class ChartDataset(Dataset):

def \_\_init\_\_(self, data\_dir, transform=None):

self.data\_dir = data\_dir

self.transform = transform

self.image\_paths = []

self.labels = []

self.class\_to\_idx = {} # To map class names (folder names) to integers

# Assuming data\_dir is like 'dataset/train', 'dataset/val', 'dataset/test'

# Each subdirectory is a class (e.g., 'uptrend', 'downtrend')

for idx, class\_name in enumerate(sorted(os.listdir(data\_dir))):

class\_path = os.path.join(data\_dir, class\_name)

if os.path.isdir(class\_path):

self.class\_to\_idx[class\_name] = idx

for img\_name in os.listdir(class\_path):

if img\_name.lower().endswith(('.png', '.jpg', '.jpeg')):

self.image\_paths.append(os.path.join(class\_path, img\_name))

self.labels.append(self.class\_to\_idx[class\_name])

self.idx\_to\_class = {v: k for k, v in self.class\_to\_idx.items()}

def \_\_len\_\_(self):

return len(self.image\_paths)

def \_\_getitem\_\_(self, idx):

img\_path = self.image\_paths[idx]

image = Image.open(img\_path).convert('RGB') # Convert to RGB to handle different source image types

label = self.labels[idx]

if self.transform:

image = self.transform(image)

return image, label

# Define transforms

# Common practice to resize to a fixed size for CNNs (e.g., 224x224 or 256x256)

train\_transforms = transforms.Compose([

transforms.Resize((224, 224)), # All images will be resized to 224x224

transforms.RandomHorizontalFlip(), # Data augmentation

transforms.RandomRotation(10), # Data augmentation

transforms.ToTensor(), # Convert PIL Image to PyTorch Tensor

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # ImageNet normalization (common)

])

val\_test\_transforms = transforms.Compose([

transforms.Resize((224, 224)),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

# For loading data in main.py:

# train\_dataset = ChartDataset(data\_dir='./dataset/train', transform=train\_transforms)

# train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

# ... similar for val and test

**2. src/model\_architecture.py** This file will define your deep learning model. Start with a simple CNN.

Python

# src/model\_architecture.py

import torch

import torch.nn as nn

import torch.nn.functional as F

from torchvision import models # For using pre-trained models

class ChartPatternClassifier(nn.Module):

def \_\_init\_\_(self, num\_classes):

super(ChartPatternClassifier, self).\_\_init\_\_()

# Using a pre-trained ResNet as a backbone (transfer learning)

# ResNet18 is a good starting point for smaller datasets

self.backbone = models.resnet18(weights=models.ResNet18\_Weights.IMAGENET1K\_V1)

# Freeze backbone layers if desired (optional for small datasets)

# for param in self.backbone.parameters():

# param.requires\_grad = False

# Replace the final classification layer

# The number of input features to this layer depends on the backbone (e.g., 512 for ResNet18)

num\_ftrs = self.backbone.fc.in\_features

self.backbone.fc = nn.Linear(num\_ftrs, num\_classes)

def forward(self, x):

return self.backbone(x)

**3. src/utils.py** This file can hold helper functions like evaluation metrics, saving/loading models, etc.

Python

# src/utils.py

import torch

from sklearn.metrics import confusion\_matrix, f1\_score, accuracy\_score, classification\_report

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

def evaluate\_model(model, dataloader, device, num\_classes, class\_names):

model.eval()

all\_labels = []

all\_preds = []

with torch.no\_grad():

for images, labels in dataloader:

images, labels = images.to(device), labels.to(device)

outputs = model(images)

\_, predicted = torch.max(outputs.data, 1)

all\_labels.extend(labels.cpu().numpy())

all\_preds.extend(predicted.cpu().numpy())

accuracy = accuracy\_score(all\_labels, all\_preds)

f1 = f1\_score(all\_labels, all\_preds, average='weighted') # 'weighted' for imbalance

print(f"Accuracy: {accuracy:.4f}")

print(f"F1-Score (weighted): {f1:.4f}")

print("\nClassification Report:")

print(classification\_report(all\_labels, all\_preds, target\_names=class\_names, zero\_division=0))

cm = confusion\_matrix(all\_labels, all\_preds)

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class\_names, yticklabels=class\_names)

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.title('Confusion Matrix')

plt.show()

def save\_model(model, path):

torch.save(model.state\_dict(), path)

print(f"Model saved to {path}")

def load\_model(model, path, device):

model.load\_state\_dict(torch.load(path, map\_location=device))

model.to(device)

model.eval()

print(f"Model loaded from {path}")

return model

**4. main.py (Full Code)** This is the main script that ties everything together.

Python

# main.py

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader

from tqdm import tqdm

import os

import matplotlib.pyplot as plt

import seaborn as sns # For plotting data distribution

# Import custom modules

from src.dataset\_loader import ChartDataset, train\_transforms, val\_test\_transforms

from src.model\_architecture import ChartPatternClassifier

from src.utils import evaluate\_model, save\_model, load\_model

# --- 0. Device Configuration ---

if torch.backends.mps.is\_available():

device = torch.device("mps")

print("Using Apple Silicon GPU (MPS) for computations.")

else:

device = torch.device("cpu")

print("Apple Silicon GPU (MPS) not available or not an M1/M2/M3 Mac. Using CPU for computations.")

# --- 1. Data Preparation and Loading ---

# Adjust these paths to where your 'trading\_chart\_dataset' is located

TRAIN\_DATA\_DIR = './dataset/trading\_chart\_dataset/train'

VAL\_DATA\_DIR = './dataset/trading\_chart\_dataset/val'

TEST\_DATA\_DIR = './dataset/trading\_chart\_dataset/test'

MODEL\_SAVE\_PATH = './trained\_model.pth'

# Create dataset instances

train\_dataset = ChartDataset(data\_dir=TRAIN\_DATA\_DIR, transform=train\_transforms)

val\_dataset = ChartDataset(data\_dir=VAL\_DATA\_DIR, transform=val\_test\_transforms)

test\_dataset = ChartDataset(data\_dir=TEST\_DATA\_DIR, transform=val\_test\_transforms)

# Get class names and number of classes from the training dataset

class\_names = list(train\_dataset.class\_to\_idx.keys())

num\_classes = len(class\_names)

print(f"Detected classes: {class\_names}")

print(f"Number of classes: {num\_classes}")

# Create DataLoader instances

BATCH\_SIZE = 16 # Adjust batch size based on your GPU memory

train\_loader = DataLoader(train\_dataset, batch\_size=BATCH\_SIZE, shuffle=True)

val\_loader = DataLoader(val\_dataset, batch\_size=BATCH\_SIZE, shuffle=False)

test\_loader = DataLoader(test\_dataset, batch\_size=BATCH\_SIZE, shuffle=False)

print(f"Train samples: {len(train\_dataset)}")

print(f"Validation samples: {len(val\_dataset)}")

print(f"Test samples: {len(test\_dataset)}")

# --- 2. Model Initialization ---

model = ChartPatternClassifier(num\_classes=num\_classes).to(device)

# --- 3. Loss Function and Optimizer ---

criterion = nn.CrossEntropyLoss() # Suitable for multi-class classification

optimizer = optim.Adam(model.parameters(), lr=0.001)

# --- 4. Training Loop ---

NUM\_EPOCHS = 20 # You might need more epochs for better performance

print("\nStarting training...")

for epoch in range(NUM\_EPOCHS):

model.train()

running\_loss = 0.0

for images, labels in tqdm(train\_loader, desc=f"Epoch {epoch+1}/{NUM\_EPOCHS} (Train)"):

images, labels = images.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item() \* images.size(0)

epoch\_loss = running\_loss / len(train\_dataset)

print(f"Epoch {epoch+1}/{NUM\_EPOCHS}, Train Loss: {epoch\_loss:.4f}")

# Validation phase

model.eval()

val\_running\_loss = 0.0

correct\_predictions = 0

total\_predictions = 0

with torch.no\_grad():

for images, labels in tqdm(val\_loader, desc=f"Epoch {epoch+1}/{NUM\_EPOCHS} (Val) "):

images, labels = images.to(device), labels.to(device)

outputs = model(images)

loss = criterion(outputs, labels)

val\_running\_loss += loss.item() \* images.size(0)

\_, predicted = torch.max(outputs.data, 1)

total\_predictions += labels.size(0)

correct\_predictions += (predicted == labels).sum().item()

val\_loss = val\_running\_loss / len(val\_dataset)

val\_accuracy = correct\_predictions / total\_predictions

print(f"Epoch {epoch+1}/{NUM\_EPOCHS}, Val Loss: {val\_loss:.4f}, Val Accuracy: {val\_accuracy:.4f}")

# Save the trained model

save\_model(model, MODEL\_SAVE\_PATH)

# --- 5. Evaluation on Test Set ---

print("\nEvaluating on test set...")

evaluate\_model(model, test\_loader, device, num\_classes, class\_names)

print("\nProject execution complete!")